Lecture 15: Sorting II, Intro Graphs

CSE 373 Data Structures and Algorithms

Administrivia

Project 2 due tonight.

Project 3 out this evening – one week project due Wednesday August 7

- P3 is a step up in difficulty compared to P1 and P2.

Exercise 3 due Friday

Administrivia

Midterm scores are on gradescope

Median 71.3% (58.5/82)
Mean 70.9% (58.1/82)
Standard Deviation 12.03% (9.86 points)

The exam was difficult (and a bit longer than I wanted it to be)

These scores are very impressive – you've learned a lot so far.

Administrivia

If you're concerned about your grade:

- Because you need [GPA X] to graduate/for your major
- Because you're trying to transfer into CSE
- Because you just want to know how you're doing

I'll have office hours 2:30-4 on Friday (CSE 330)

Just for one-on-one discussions of where you stand in the class and what you should expect to need to do in the second half of the course.

TAs will have regular office hours then (at the breakouts) for content questions.

You can also email me for appointments.

Algorithm Design Patterns

Algorithms don't just come out of thin air.

There are common patterns we use to design new algorithms.

Many of them are applicable to sorting (we'll see more patterns later in the quarter)

Invariants/Iterative improvement
- Step-by-step make one more part of the input your desired output.

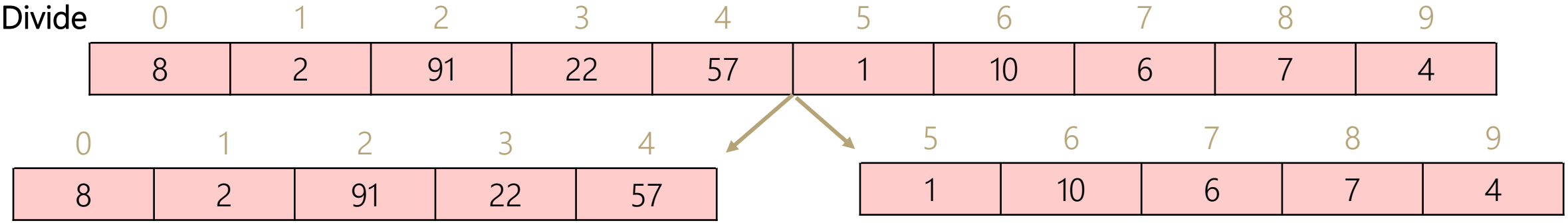
Using data structures
- Speed up our existing ideas

Divide and conquer
- Split your input
- Solve each part (recursively)
- Combine solved parts into a single

Merge Sort

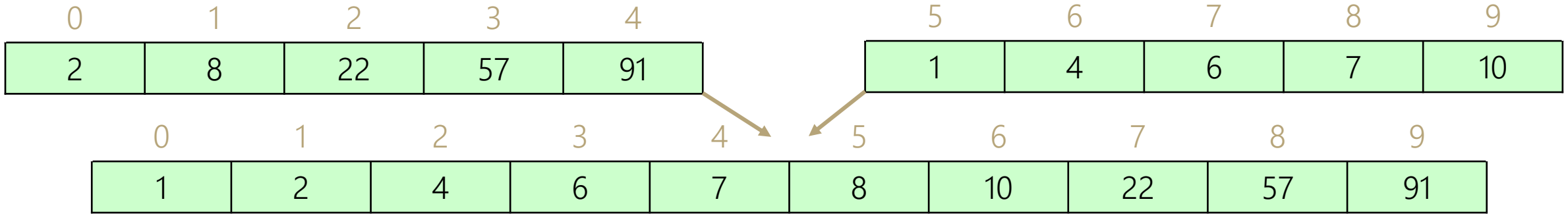
https://www.youtube.com/watch?v=XaqR3G_NVoo

Divide



Sort the pieces through the **magic** of recursion

Combine



Merge Sort

```
mergeSort(input) {
   if (input.length == 1)
      return
   else
      leftHalf = mergeSort(new [0, ..., mid])
      rightHalf = mergeSort(new [mid + 1, ...])
      return merge(leftHalf, rightHalf)
}
```

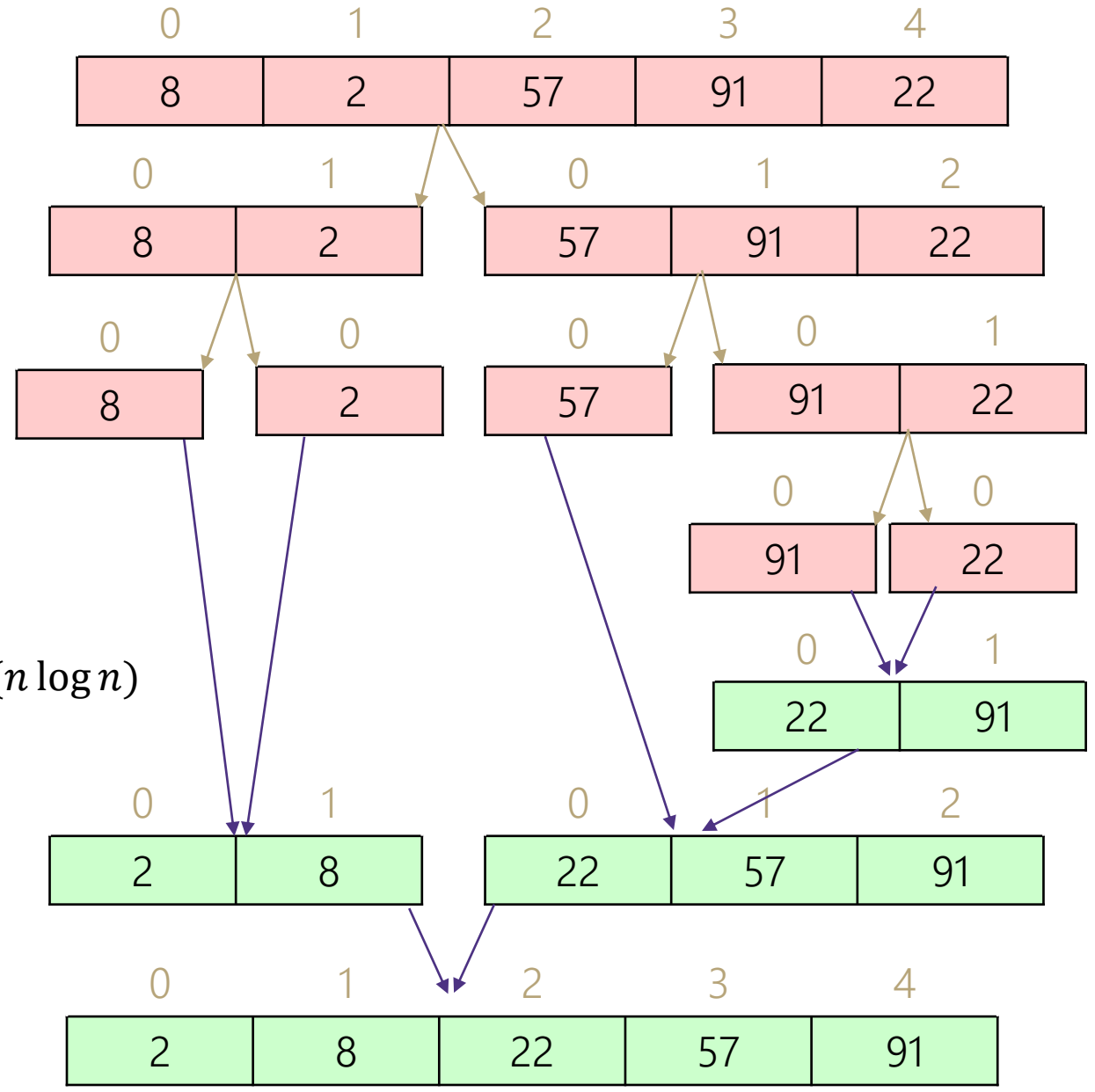
Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases} = O(n \log n)$

Best case runtime? Same

Average runtime? Same

Stable? Yes

In-place? No



Divide and Conquer

There's more than one way to divide!

Mergesort:

Split into two arrays.

- Elements that just happened to be on the left and that happened to be on the right.

Quicksort:

Split into two arrays.

- Elements that are "small" and elements that are "large"
- What do I mean by "small" and "large" ?

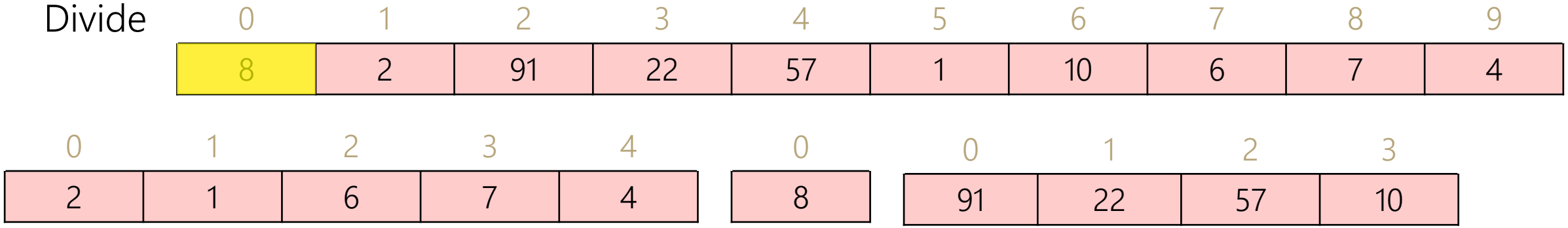
Choose a "pivot" value (an element of the array)

One array has elements smaller than pivot, other has elements larger than pivot.

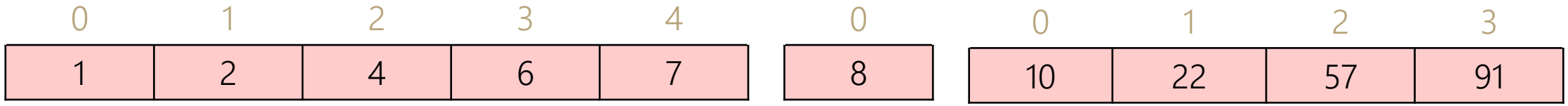
Quick Sort v1

https://www.youtube.com/watch?v=ywWBy6J5gz8

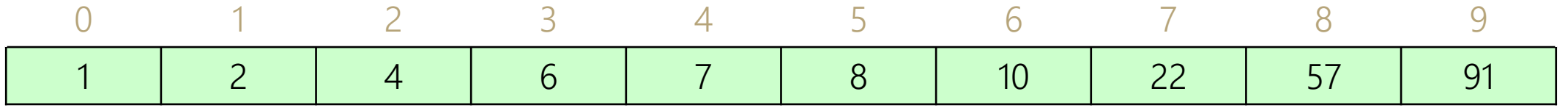
Divide



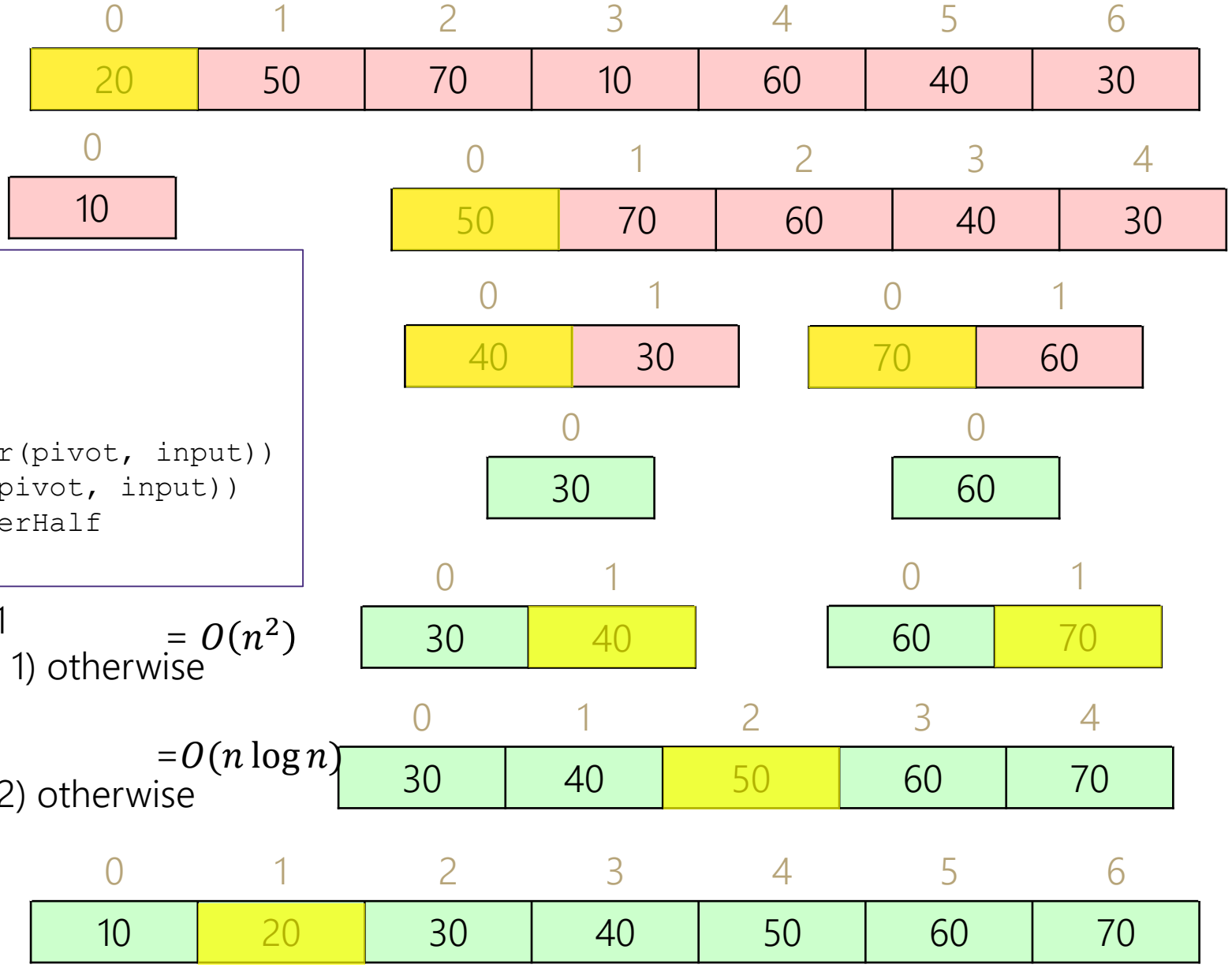
Sort the pieces through the **magic** of recursion



Combine (no extra work if in-place)



Quick Sort v1




```
quickSort(input) {
   if (input.length == 1)
      return
   else
      pivot = getPivot(input)
      smallerHalf = quickSort(getSmaller(pivot, input))
      largerHalf = quickSort(getBigger(pivot, input))
      return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + T(n-1) \text{ otherwise} \end{cases}$ $= O(n^2)$

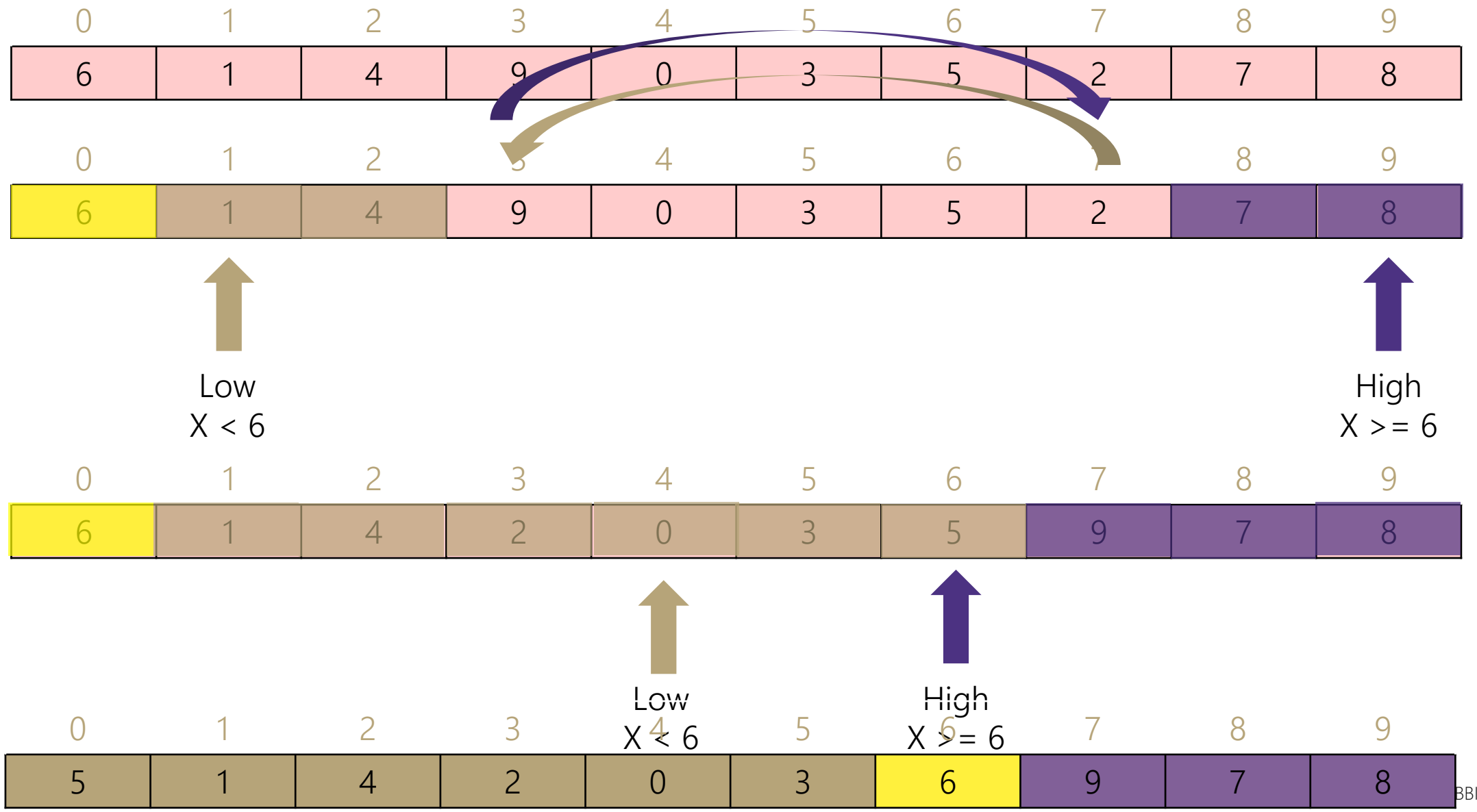
Best case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + 2T(n/2) \text{ otherwise} \end{cases}$ $= O(n \log n)$

In-practice runtime? Just trust me $O(n \log n)$

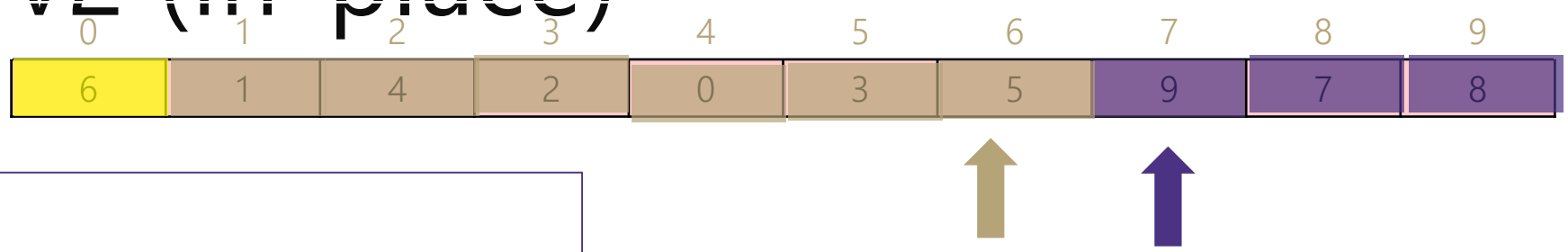
Stable? No

In-place? Can be done

Quick Sort v2 (swapping in-place)



Quick Sort v2 (in-place)




```
quickSort(input) {
   if (input.length == 1)
      return
   else
      pivot = getPivot(input)
      smallerHalf = quickSort(getSmaller(pivot, input))
      largerHalf = quickSort(getBigger(pivot, input))
      return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + T(n-1) \text{ otherwise} \end{cases}$ = O(n^2)

Best case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + 2T(n/2) \text{ otherwise} \end{cases}$ = O(nlogn)

In-practice runtime? Just trust me = O(nlogn)

Stable? No

In-place? Yes

Can we do better?

We'd really like to avoid hitting the worst case.

Key to getting a good running time, is always cutting the array (about) in half.

How do we choose a good pivot?

Here are four options for finding a pivot. What are the tradeoffs?

- Just take the first element
- Take the median of the first, last, and middle element
- Take the median of the full array
- Pick a random element as a pivot

Pivots

Just take the first element

- fast to find a pivot
- But (e.g.) nearly sorted lists get $\Omega(n^2)$ behavior overall

Take the median of the first, last, and middle element

- Guaranteed to not have the absolute smallest value.
- On real data, this works quite well…
- But worst case is still $\Omega(n^2)$

Median of three is a common choice in practice

Take the median of the full array

- Can actually find the median in $O(n)$ time (google Median of Meidans). It's **complicated.**
- $O(n\ log\ n)$ even in the worst case….but the constant factors are **awful**. No one does quicksort this way.

Pick a random element as a pivot

- somewhat slow constant factors
- Get $O(n \log n)$ running time with probability at least $1 - 1/n^2$
- "adversaries" can't make it more likely that we hit the worst case.

Worst Case

There's a pattern you might have noticed:

The worst case for our sorting algorithm has never gotten better than $\Theta(n \log n)$

Is there a $\Theta(n)$ worst-case algorithm out there?

No!

Comparison Sorting Lower Bound

Any sorting algorithm which only interacts with its input by comparing elements must take $\Omega(\mathrm{n} \log \mathrm{n})$ time in the worst case.

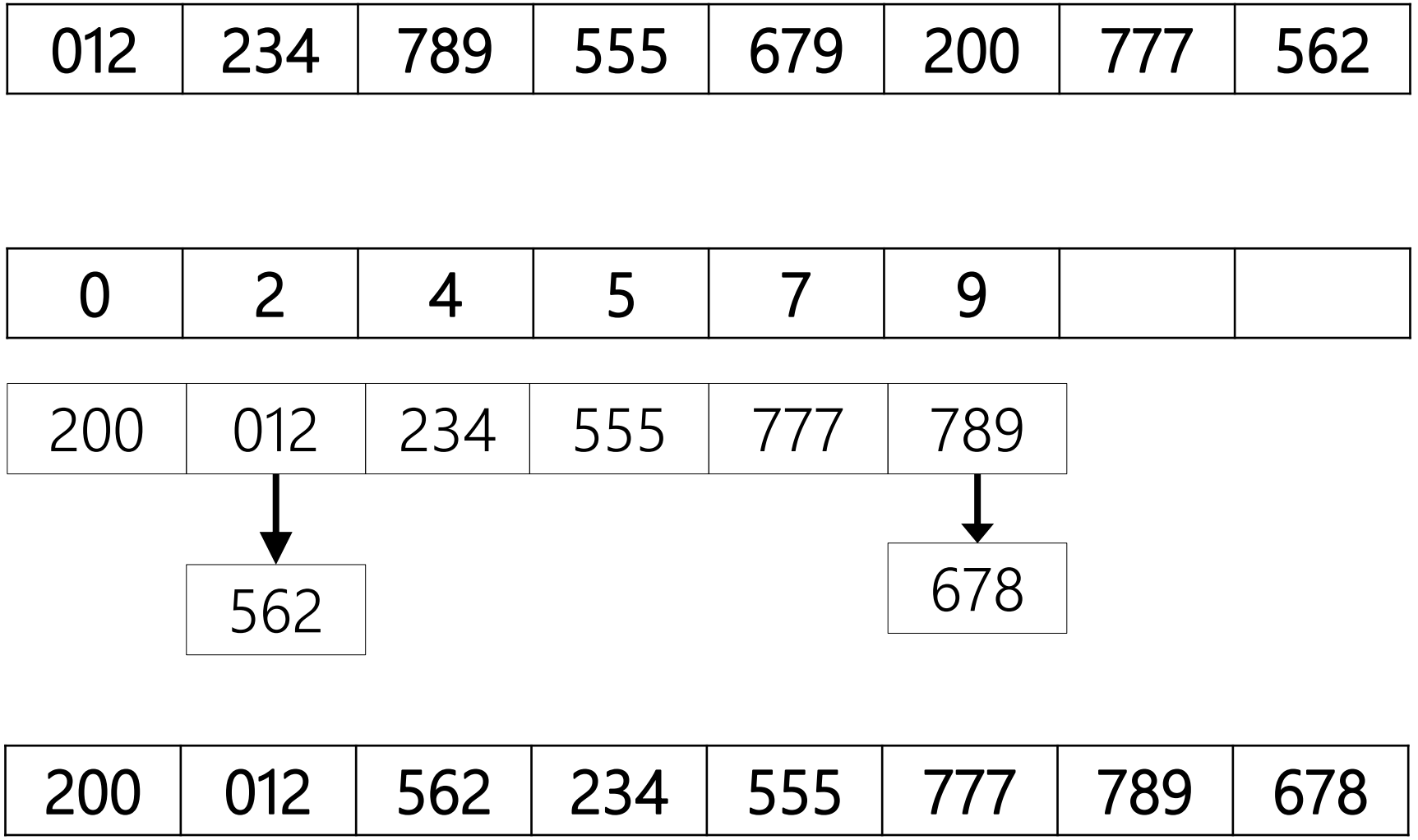
Radix Sort

For each “digit” (starting at the “ones place”)

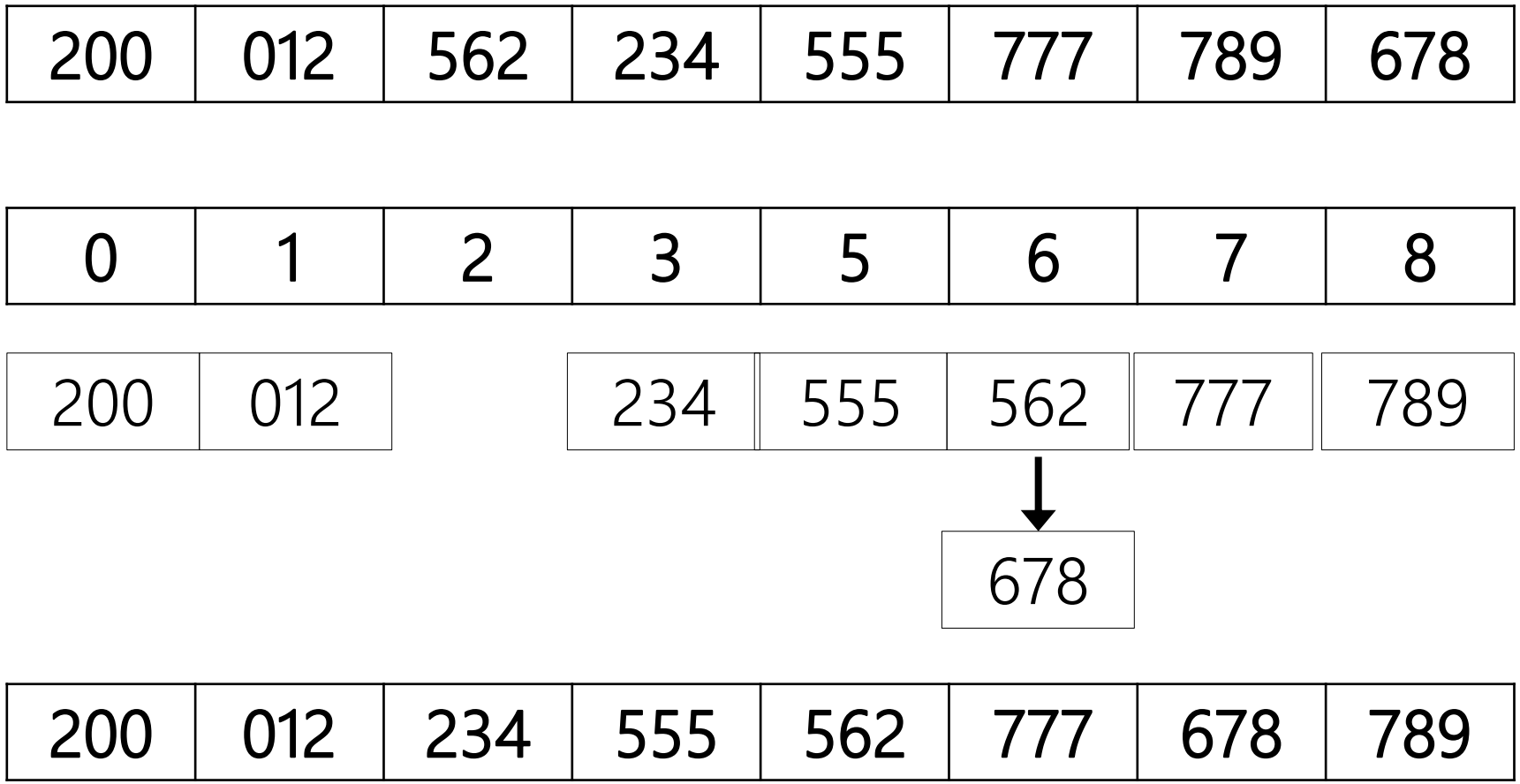
- Run a “bucket sort” with respect to that digit

- I.e. make an array, where each index corresponds to one of the possible values
- Place the next element you see at the **end** of the list of elements in that bucket.
- Keep the sort stable!

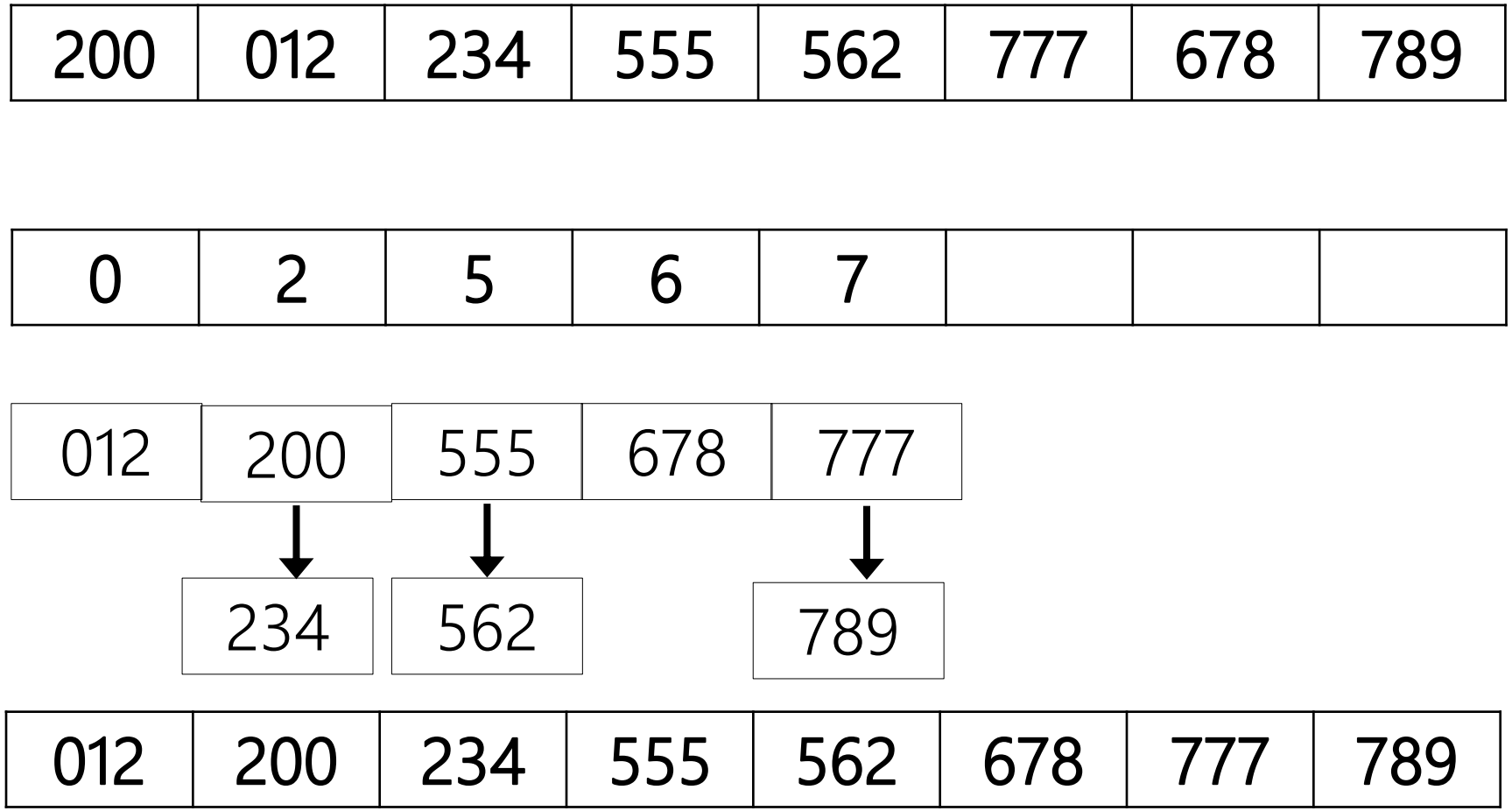
Radix Sort: Ones Place



Radix Sort: Tens Place



Radix Sort: Hundreds Place



Radix Sort

Key idea: by keeping the sorts stable, when we sort by the hundreds place, ties are broken by tens place (then by ones place).

Running time? $O((n + r)d)$

Where d is number of digits in each entry,

r is the radix, i.e. the base of the number system.

If you're sorting base-10 numbers, r is $\mathbf{10}$.

If you're sorting all lower-case English words, r is 26.

Radix Sort

When can you use it?

ints and strings. As long as they aren’t too large.

But you have to know you’re sorting ints and strings.

Summary

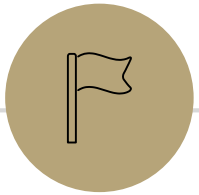
You have a bunch of data. How do you sort it?

Honestly…use your language's default implementation

- It's been carefully optimized.

Unless you really know something about your data, or the situation your in

- Not a lot of extra memory? Use an in place sort.
- Want to sort repeatedly to break ties? Use a stable sort.
- Know your data all falls into a small range? Maybe radix sort.



Graphs

ADTs so far

Queues and Stacks

- We want to process our data in some order (based on when they were inserted)

Lists

- We want to maintain an order, but add or remove from anywhere

Priority Queues

- Our data had some priority we needed to keep track of, and wanted to process in order of importance.

Dictionaries

- Our data points came as (key, value) pairs.
- Quickly find the value for a key

Graphs

Secure | https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

Not logged in Talk Contributions Create account Log in

WIKIPEDIA
The Free Encyclopedia

Article Talk | Read Edit View history | Search Wikipedia

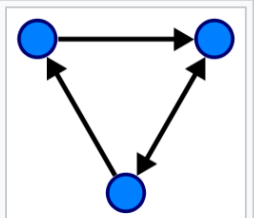
Graph [redacted]

From Wikipedia, the free encyclopedia

In computer science, a **graph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics, specifically the field of graph theory.

A graph data structure consists of a finite (and possibly mutable) set of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some *edge value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).



A graph with three vertices and three edges.

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Contents [hide]

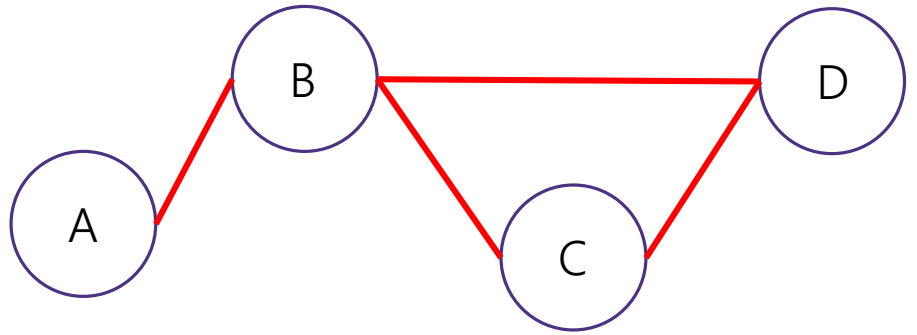
1 Operations

We'll list Graphs as one of our ADTs…
But don't let that limit your thinking. They are more versatile than any ADT we've seen before.

Graphs

Represent data points and the relationships between them.

That's vague.



Formally:

A graph is a pair: G = (V,E)

V: set of **vertices** (aka **nodes**) $\{A, B, C, D\}$

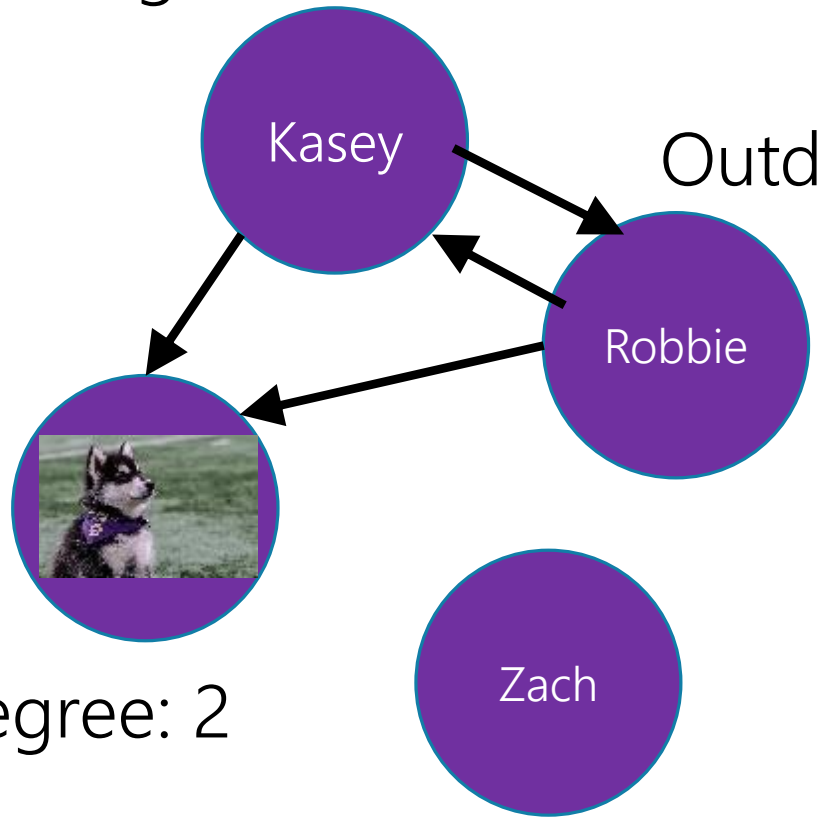
E: set of **edges** $\{(A, B), (B, C), (B, D), (C, D)\}$

- Each edge is a pair of vertices.

Graph Terms

Graphs can be directed or undirected.

Following on twitter.

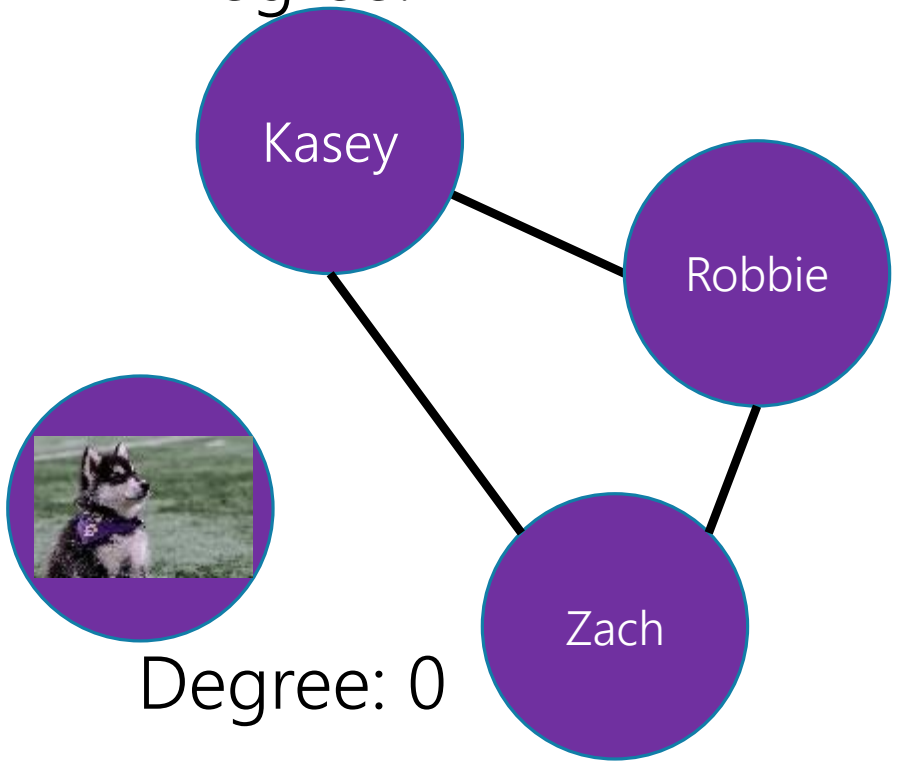


Outdegree: 2

Indegree: 2

This graph is **disconnected**.

Degree: 2



Degree: 0

Friendships on Facebook.

Making Graphs

If your problem has **data** and **relationships**, you might want to represent it as a graph

How do you choose a representation?

Usually:

Think about what your "fundamental" objects are

- Those become your vertices.

Then think about how they're related

- Those become your edges.

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

Facebook friendships

Input data for the “6 degrees of Kevin Bacon” game

Course Prerequisites

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

- Vertices: webpages. Edges from a to b if a has a hyperlink to b.

Facebook friendships

- Vertices: people. Edges: if two people are friends

Input data for the “6 Degrees of Kevin Bacon” game

- Vertices: actors. Edges: if two people appeared in the same movie
- Or: Vertices for actors and movies, edge from actors to movies they appeared in.

Course Prerequisites

- Vertices: courses. Edge: from a to b if a is a prereq for b.